CPEN 311: Digital Systems Design

Bit width mismatches in Verilog

1

Bit width mismatches

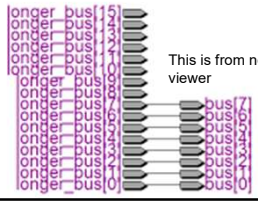
The crux of the problem:

```
wire [7:0] bus;
wire [15:0] longer_bus;

assign bus = longer_bus;
```

Verilog lets you do this! The compiler gives a warning, but you might miss it:

truncated value with size 16 to match size of target (8)



This is from netlist viewer

2

Verilog rule: if the destination is shorter than the source, the least significant bits of the source are used.

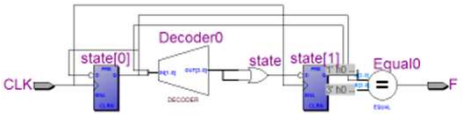
3

Beware of tricky errors:

```
reg [1:0] state;

always_ff @(posedge CLK)
    case (state)
       3'b000:  state <= 3'b001;
       3'b001:  state <= 3'b010;
       3'b010:  state <= 3'b011;
       3'b011:  state <= 3'b100;
       3'b100:  state <= 3'b101;
       default: state <= 3'b000;
   endcase
```

This will go back to state 00 !!



4

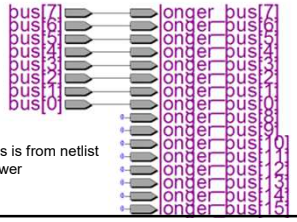
Bit width mismatches

The other direction:

```
wire [7:0] bus;
wire [15:0] longer_bus;

assign longer_bus = bus;
```

Verilog lets you do this! No warning message! Extra high-order bits are assigned 0.



This is from netlist viewer

5

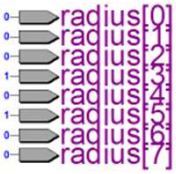
What does this do?

```
wire [7:0] radius;

assign radius = 40;
```

A constant is assumed to be 32 bits

So, this is the same as assigning a long bus to a short bus.
Radius gets its value from the lower order 8 bits of the 32-bit constant 40.



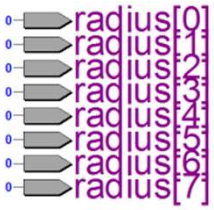
6

What does this do?

```
wire [7:0] radius;

assign radius = 256;
```

The number 256 in binary is 1_0000_0000.
Extract the 8 order low bits, and a 0 gets assigned to radius!



7

Is this a bitwidth mismatch?

```
wire [7:0] offsety;

assign offsety = offsety+1;
```

Rule: if you add two numbers, the bitwidth of the result is the maximum width of the two inputs.

So, this this case, the constant 1 is treated as 32 bits wide

So the result is 32 bits wide.

You are assigning it to an 8 bit bus, so you should expect a warning

But, it does work as you think, it just truncates the upper 24 bits.

8

Is this a bitwidth mismatch?

```
wire [7:0] offsety;

assign offsety = offsety + 1'b1;
```

In this case, there is theoretically no bitwidth mismatch, since the result is 8 bits wide, and the tools let you do this

(But, really, if it were civilized it would scream at you that 8'b1 + 1'b1 is nonsense)

9

Sign Extension

Actually, it is a bit more complicated than this:

For an addition:
- Each operand is extended to the width of the largest (including the left-hand side)
- Addition is performed
- Result is truncated to fit into the result

If all operands are unsigned, **zero extension** is done by padding with 0

10

Sign Extension

```
wire [3:0] A, B;
wire [4:0] SUM;

assign SUM = A + B;
```

In this case, A and B are **not signed,** so they are **zero-extended** to 5 bits by including a 0 in the MSB

The result is assigned to SUM which is a 5 bit register (allows for carry)

11

Sign Extension

Tricky:

If **all** right-hand expressions are **signed**,
then the result is **sign-extended**:
the sign bit is replicated left as many bits as req'd

If **any** right-hand expression is **unsigned**,
then the result is **zero-extended**:
0 padding for **all** operands (even the signed ones!)

12

Getting rid of warnings:

```
wire [7:0] bus;
wire [15:0] longer_bus;

assign bus = longer_bus[7:0];
```

More explicit, probably always a good idea.

13

Getting rid of warnings:

```
wire [7:0] radius;

assign radius = 8'd40;
```

```
wire [7:0] offsety;

assign offsety = offsety + 8'b1;
```

More explicit, probably always a good idea.

14

defparam

You can define/use parameters like this:

```
module adder_b (sum, cout, a, b);
parameter width = 16;
input [width-1:0] a;
input [width-1:0] b;
output [width-1:0] sum;
output cout;

assign {cout,sum} = a + b;
endmodule
```

{A,B} is the **concatenation** operator
Makes a new, wider "bus" by putting bits from A in the MSB position, bits of B in the LSB position

15

defparam

You can over-ride parameters like this:

Top Level

```
adder_b u0 (sum, cout, a, b);
defparam u0.width = 8;
```

Module being instantiated

```
module adder_b (sum, cout, a, b);
parameter width = 16;
input [width-1:0] a;
input [width-1:0] b;
output [width-1:0] sum;
output cout;

assign {cout,sum} = a + b;
endmodule
```

This would make an 8-bit adder

16

defparam

Short-cut:

Top Level

```
adder_b #(8) u0 (sum, cout, a, b);
```

Module being instantiated

```
module adder_b (sum, cout, a, b);
parameter width = 16;
input [width-1:0] a;
input [width-1:0] b;
output [width-1:0] sum;
output cout;

assign {cout,sum} = a + b;
endmodule
```

This would make an 8-bit adder

17

Are these the same thing? Does either give a warning?

```
parameter WIDTH = 3;

wire [WIDTH-1:0] r = 1'b0;
```

```
parameter WIDTH = 3;

wire [WIDTH-1:0] r = {WIDTH{1'b0}};
```

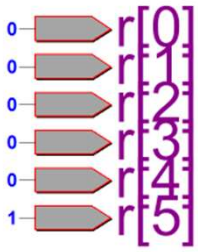
{N{A}} is the **replication** operator
Makes a new, wider result by concatenating N copies of A together

18

What does this do?

```
parameter WIDTH = 6;

wire [WIDTH-1:0] r = {1'b1, {WIDTH-1{1'b0} }};
```



19

Summary

It is important to always be aware of bit-width mismatches.

Ideally, you will write code that has none. Then, if you see a bit-width mismatch warning, you will know something is wrong.

When you are debugging, these bit-width mismatch warnings would be a good place to start.

20